US009104427B2

(12) **United States Patent**
Heller, Jr.

(10) **Patent No.: US 9,104,427 B2**
(45) **Date of Patent: Aug. 11, 2015**

(54) **COMPUTING SYSTEM WITH TRANSACTIONAL MEMORY USING MILLICODE ASSISTS**

(71) Applicant: **International Business Machines Corporation**, Armonk, NY (US)

(72) Inventor: **Thomas J. Heller, Jr.**, Rhinebeck, NY (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 388 days.

(21) Appl. No.: **13/655,636**

(22) Filed: **Oct. 19, 2012**

(65) **Prior Publication Data**

US 2013/0042094 A1 Feb. 14, 2013

Related U.S. Application Data

(60) Division of application No. 11/928,533, filed on Oct. 30, 2007, which is a continuation-in-part of application No. 11/748,044, filed on May 14, 2007, now Pat. No. 8,321,637.

(51) **Int. Cl.**
G06F 9/46 (2006.01)
G06F 9/38 (2006.01)
(Continued)

(52) **U.S. Cl.**
CPC ***G06F 9/3885*** (2013.01); ***G06F 9/3004*** (2013.01); ***G06F 9/3017*** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC G06F 9/467
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,686,641 A 8/1972 Logan et al.
5,428,761 A 6/1995 Herlihy et al.
(Continued)

FOREIGN PATENT DOCUMENTS

EP 1643371 A2 4/2006
JP 2006107494 4/2006
WO WO2007067390 A2 6/2007

OTHER PUBLICATIONS

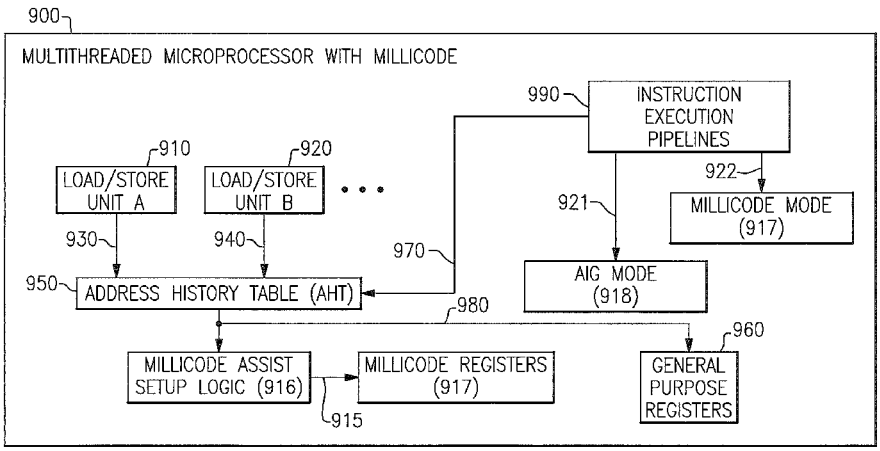
Heller, L. C., Farrell, M. S. "Millicode in an IBM zSeries processor" IBM J. Res. & Dev. vol. 48 No. 3/4 May/Jul. 2004.*
(Continued)

Primary Examiner — Jacob A Petranek
(74) *Attorney, Agent, or Firm* — Cantor Colburn LLP; Margaret McNamara

(57) **ABSTRACT**

A computing system processes memory transactions for parallel processing of multiple threads of execution with millicode assists. The computing system transactional memory support provides a Transaction Table in memory and a method of fast detection of potential conflicts between multiple transactions. Special instructions may mark the boundaries of a transaction and identify memory locations applicable to a transaction. A 'private to transaction' (PTRAN) tag, directly addressable as part of the main data storage memory location, enables a quick detection of potential conflicts with other transactions that are concurrently executing on another thread of said computing system. The tag indicates whether (or not) a data entry in memory is part of a speculative memory state of an uncommitted transaction that is currently active in the system. Program millicode provides transactional memory functions including creating and updating transaction tables, committing transactions and controlling the rollback of transactions which fail.

11 Claims, 10 Drawing Sheets



(51) **Int. Cl.**
G06F 9/30 (2006.01)
G06F 11/14 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F9/30087*** (2013.01); ***G06F 9/3834*** (2013.01); ***G06F 9/3842*** (2013.01); ***G06F 9/3851*** (2013.01); ***G06F 9/466*** (2013.01); ***G06F 9/467*** (2013.01); ***G06F 11/1405*** (2013.01); ***G06F 11/1474*** (2013.01)

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,438,761	A		8/1995	Krumszyn et al.
5,553,291	A		9/1996	Tanaka et al.
5,701,432	A		12/1997	Wong et al.
5,742,785	A		4/1998	Stone et al.
5,946,711	A		8/1999	Donnelly
5,963,922	A		10/1999	Helmering
5,974,438	A		10/1999	Neufeld
6,035,379	A		3/2000	Raju et al.
6,052,695	A		4/2000	Abe et al.
6,067,617	A	*	5/2000	Webb et al. 712/245
6,078,999	A		6/2000	Raju et al.
6,108,776	A	*	8/2000	Check et al. 712/240
6,360,220	B1		3/2002	Forin
6,360,231	B1		3/2002	Pong et al.
6,381,676	B2		4/2002	Aglietti et al.
6,510,498	B1		1/2003	Holzie et al.
6,611,906	B1		8/2003	McAllister et al.
6,651,146	B1		11/2003	Srinivas et al.
6,738,837	B1		5/2004	Wyland
6,748,505	B1		6/2004	Dakhil
6,826,757	B2		11/2004	Steele, Jr. et al.
6,862,664	B2		3/2005	Tremblay et al.
6,874,065	B1		3/2005	Pong et al.
6,880,045	B2		4/2005	Pong et al.
6,880,071	B2		4/2005	Steele, Jr. et al.
6,938,130	B2		8/2005	Jacobson et al.
6,981,110	B1		12/2005	Melvin
7,000,234	B1		2/2006	Shavit et al.
7,017,160	B2		3/2006	Martin et al.
7,039,794	B2		5/2006	Rodgers et al.
7,089,374	B2		8/2006	Tremblay et al.
7,107,402	B1		9/2006	Melvin
7,117,502	B1		10/2006	Harris
7,350,034	B2		3/2008	Shen
7,421,544	B1	*	9/2008	Wright et al. 711/150
7,536,517	B2		5/2009	Harris
7,689,788	B2		3/2010	Moir et al.
7,730,286	B2		6/2010	Petersen et al.
7,865,701	B1		1/2011	Tene et al.
8,095,741	B2		1/2012	Heller, Jr. et al.
8,095,750	B2		1/2012	Heller, Jr.
8,117,403	B2		2/2012	Heller, Jr. et al.
8,321,637	B2		11/2012	Baum et al.
2002/0072071	A1		6/2002	Kientsch-Engel et al.
2002/0073071	A1		6/2002	Pong et al.
2002/0078308	A1		6/2002	Altman et al.
2002/0103804	A1		8/2002	Rothschild et al.
2002/0143847	A1		10/2002	Smith
2002/0161815	A1		10/2002	Bischof et al.
2002/0199069	A1		12/2002	Joseph
2003/0066056	A1		4/2003	Petersen et al.
2003/0079094	A1		4/2003	Rajwar et al.
2003/0084038	A1		5/2003	Balogh et al.
2003/0204682	A1		10/2003	Ueno
2004/0015642	A1		1/2004	Moir et al.
2004/0034673	A1		2/2004	Moir et al.
2004/0152948	A1		8/2004	Kim
2004/0162948	A1		8/2004	Tremblay et al.
2004/0187115	A1		9/2004	Tremblay et al.
2004/0187116	A1		9/2004	Tremblay et al.
2004/0187127	A1		9/2004	Gondi et al.
2004/0267828	A1		12/2004	Zwilling et al.
2005/0060559	A1		3/2005	McKenney
2005/0086446	A1		4/2005	McKenney et al.
2005/0097296	A1		5/2005	Chamberlain et al.
2005/0131947	A1		6/2005	Laub et al.
2005/0138298	A1		6/2005	Downer
2005/0216625	A1		9/2005	Smith et al.
2006/0085588	A1		4/2006	Rajwar et al.
2006/0085591	A1		4/2006	Kumar et al.
2006/0173885	A1		8/2006	Moir et al.
2006/0200632	A1		9/2006	Tremblay et al.
2006/0206692	A1		9/2006	Jensen
2006/0212456	A1		9/2006	Earhart
2006/0282476	A1		12/2006	Dolby et al.
2006/0288173	A1		12/2006	Shen
2006/0294326	A1		12/2006	Jacobson et al.
2007/0028056	A1		2/2007	Harris
2007/0028058	A1		2/2007	Bradley et al.
2007/0186056	A1		8/2007	Saha et al.
2007/0186215	A1	*	8/2007	Rajwar et al. 718/102
2007/0239942	A1		10/2007	Rajwar et al.
2007/0282838	A1	*	12/2007	Shavit et al. 707/8
2007/0300238	A1		12/2007	Kontothanassis et al.
2008/0022054	A1		1/2008	Hertzberg et al.
2008/0065864	A1		3/2008	Akkary et al.
2008/0098181	A1		4/2008	Moir et al.
2008/0288238	A1		11/2008	Heller, Jr.
2008/0288727	A1		11/2008	Baum et al.
2008/0288730	A1		11/2008	Heller, Jr. et al.
2008/0288819	A1		11/2008	Heller, Jr.
2009/0172292	A1		7/2009	Saha et al.
2011/0055483	A1		3/2011	Heller, Jr.
2013/0046937	A1		2/2013	Heller, Jr.

OTHER PUBLICATIONS

McDonald, Austen et al. "Architectural Semantics for Practical Transactional Memory" International Symposium on Computer Architecture, 2006.*

Moore, Kevin E. et al. "LogTM: Log-based Transactional Memory" 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12) Austin, TX Feb. 11-15, 2006.*

USPTO Communication, U.S. Appl. No. 11/928,857—Notice of Allowance, Date mailed: Oct. 24, 2013, 22 pages.

Annanian; "Unbound Transactional Memory"; Dec. 31, 2005; IEEE; 11th International Symposium on HPCA; 12 pages.

U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928,533 dated Oct. 22, 2012.

U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928,857 dated Aug. 3, 2012.

U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 12/550,844 dated Dec. 26, 2012.

U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,533 dated May 9, 2012.

U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,857 dated Dec. 27, 2011.

U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 13/656,778 dated Feb. 22, 2013.

U.S.P.T.O. Communication, Notice of Allowance for U.S. Appl. No. 11/748,044 mailed Jul. 19, 2012.

U.S.P.T.O. Communication, Notice of Allowance for U.S. Appl. No. 11/928,758 mailed Jun. 30, 2011.

"Programming in the Age of Concurrency: Software: Software Transactional Memory", Sep. 1, 2006; http://channel9.msdn.com/Showpost.aspx?postid=231495; pp. 1-7.

Rajwar et al.; "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution"; 2001; IEEE; pp. 294-305.

Rajwar et al.; "Virtualizing Transactional Memory"; Proceedings of 32nd International Symposium on Computer Architecture; IEEE; Jun. 4-8, 2005; 12 pages.

Roetter, Alex; "Writing Multithreaded JAVA Applications"; Feb. 1, 2001; pp. 1-7; retrieved from the internet Feb. 1, 2001; http://www-128.ibm.com/developworks/java/library/j-thread.html.

U.S.P.T.O. Communication, Requirement for Restriction/Election for U.S. Appl. No. 11/928,533 dated Jun. 28, 2010 .

Saha et al. "Architectural Support for Software Transactional Memory"; 39th Annual IEEE/ACM International Symposium on Microarchitecture; Dec. 9, 2006; Dec. 13, 2006; pp. 1-12.

(56) **References Cited**

OTHER PUBLICATIONS

Saha et al. McRT-STM—A High Performance Software Transactional Memory System for a Multi-Core Runitime; PPoPP '06; ACM; Mar. 29-31, 2006; pp. 187-197.
Shavit et al.; "Software Transactional Memory", Annual ACM Symposium on Principles of Distributed Computing, Proceedings of Fourteenth Annual ACM Symposium on Principles of Distributed Computing, ACM; 1995; pp. 0-33.
Subject Matter Eligibility Test (M-OR-T) for Process Claims; USPTO; pp. 1/1; Aug. 2009.
Shriraman et al.; "Hardware Acceleration of Software Transactional Memory"; May 18, 2006; pp. 1-10.
Soltis; "Inside the AS/400"; 2nd Edition; Duke Press; 1997; pp. 1-6.
"THREAD: (Computer Science)" Wikipedia; htt://en.wikipedia.org/wiki/Thread_(computer_science); Feb. 2, 2007; pp. 1-7.
"Tuple Space"; http://c2.com/cgi/wiki?TupleSpace; Sep. 20, 2006; pp. 1-5.
Wilson et al.; The Case for Compressed Caching in Virtual Memory Systems; Proceedings of the 1999 USENIX Annual Technical Conference; 1999; pp. 1-16.
Wheeler "Secure Programmer: Prevent Race Conditions"; http://www-128.ibm.com/developerworks.linux/library/1-sprace.html; Oct. 7, 2004; pp. 1-10.
Wood; "Transactional Memory—An Overview of Hardware Alternatives"; Transactional Memory Workshop; Apr. 8, 2005; pp. 1-22.
Yen et al.; LogTM-SE Decoupling Hardware Transactional Memory from Caches; Feb. 10-14, 2007; 13th Annual International Symposium on High Performance Computer Architecture; 12 pages.
Hennessy et al; Advantage Set Associativity; "Computer Architecture: a Quantitative Approach", 3rd Edition; pp. 429; 2006.
Ananian et al.; "Unbound Transactional Memory"; Research Abstracts 2006; CSAIL Publications and Digital Archive; pp. 1-4.
Banatre et al.; "Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor"; IEEE; 1990; pp. 1-8.
Chang et al.; "801 Storage: Architecture and Programming"; Journal ACM Transactions on Computer Systems (TOCS); vol. 6, Issue 1; Feb., 1988; pp. 28-50.
Chung et al.; "Tradeoffs in Transactional Memory Virtualization"; ASPLOS '06; Oct. 21-25, 2006; pp. 1-12.
Damron et al.; "Hybrid Transactional Memory"; ASPLOS '06; Sun Microsystems, Inc.; Oct. 21-25, 2006.
Dice et al., "Transactional Locking II"; 2006; pp. 1-15.
Howe, Denis; "Definition of Set Associative Cache"; FOLDOC.org; Oct. 18, 2004; pp. 1/1.
Howe, Denis; "Definition of Virtual Memory"; FOLDOC.org; Nov. 26, 2002; pp. 1/1.
Grinberg et al.; "Investigation of Transactional Memory Using FPGAs"; School of Electrical Engineering; Tel Aviv University; Tel Aviv, Israel; Feb. 2006; pp. 1-4.
U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/748,044 dated Nov. 10, 2009.
U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928.533 dated Apr. 8, 2011.
U.S.P.T.O. Communication, Final Rjection for U.S. Appl. No. 11/928,594 dated Jul. 9, 2010.
U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928,661 dated Aug. 27, 2010.
U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928,758 dated Jul. 9, 2010.
U.S.P.T.O. Communication, Final Rejection for U.S. Appl. No. 11/928,857 dated Jun. 8, 2011.
Hammond et al.; "Transactional Memory Coherence and Consistency"; Stanford University; 2004; pp. 1-12.
Hapner et al.; "Java Message Service", Version 1.1, Sun Microsystems; Apr. 12, 2002.
Harris et al.; "Language Support for Lightweight Transactions", ACM SIGPLAN Notices, vol. 38, No. 11, ACM; Nov. 2003; pp. 1-15.
Harris et al.; "Transactional Memory with Data Invariants"; 2006; pp. 1-12.
Heller, L. C. et al.; "Millicode in an IBM zSeries Processor"; IBM Journal of Research and Development, vol. 48, No. 3/4, May/Jul. 2004; pp. 425-434.
Herlihy; "What Can We Prove About Transactional Memory?" Aug. 21-22, 2006; Brown University; pp. 1-67.
Herlihy et al.; "Transactional Memory: Architectural Support for Lock-Free Data Structures"; IEEE; 1993; pp. 289-300.
Java.lang Class Thread; Sun Microsystems; Apr. 5, 2003; retrieved from the internet Apr. 2, 2010; <http://web.archive.org/web/20040207134551/http://java.sun.com/j2se/1.5.0/docs/api/java/lang>; pp. 1-28.
Katz et al.; "Implementing a Cache Consistency Protocol, Research Paper"; Electrical Engineering and Computer Science Department; University of California Berkeley; Oct., 1984; pp. 1-31.
Kerns, Tamra; "The Advantages of Multithreaded Applications"; copyright 1998.
Kongetira et al.; "Niagara: A 32-Way Multithreaded Sparc Processor"; IEEE; 2005; pp. 21-29.
Kuszmaul et al.; "Transactions Everywhere" http://www.cs.wfsc.edu/-rajwar/tm-workshop/position_statements; htm pp. 1-19; 2003.
Lee, "The Problem with Threads", Innovative Technology for Computing Professionals, May 2006; pp. 1-19.
MATE: Micro Assist Thread Engine; IPCOM000027405D; IBM; Apr. 7, 2004.
McDonald et al.; "Architectural Semantics for Practical Transactional Memory"; Proceedings of the 33rd International Symposium on Computer Architecture; IEEE; Jun. 17-21, 2006; pp. 1-12.
Moore, K.E., et al.; LogTM: Log-based Transactional Memory. Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12); IEEE; Feb. 11-15, 2006.
Mohamed; "The Posix Interface for the Unicon Programming Language"; 1997; http://www.drones.com/unicon; pp. 1-21.
"Multithreaded Programming Guide Products and Services"; Nov. 2, 2010; http://docsun.cites,uiuc.edu/sun_docs/C/solaris_9/SUNWdev/MTP/p56.html; pp. 1-4.
"Multithreaded Programming Guide"; 2008; http://docs.sun.com/app.docs/doc/806-6867/6jfpgdcop?1=sv&a=view; pp. 1-13; Sun Microsystems Inc.; 1994.
"Native POSIX Thread Library", http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library; 2010; pp. 1-3.
U.S.P.T.O. Communication, Notice of Allowance for U.S. Appl. No. 11/928,661 mailed May 18, 2011.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/748,044 dated Apr. 13, 2011.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/748,044 dated Apr. 14, 2009.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,661 dated Apr. 2, 2010.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,758 dated Apr. 16, 2010.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,857 dated Nov. 26, 2010.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928.533 dated Oct. 29, 2010.
U.S.P.T.O. Communication, Non-Final Rejection for U.S. Appl. No. 11/928,594 dated Mar. 8, 2010.
U.S.P.T.O. Communication, Notice of Allowance for U.S. Appl. No. 11/928,594 mailed May 25, 2011.
PCT/EP2010/062302—International Search Report dated Nov. 22, 2010.
PCT/EP2010/062302—Written Opinion dated Nov. 22, 2010.

* cited by examiner

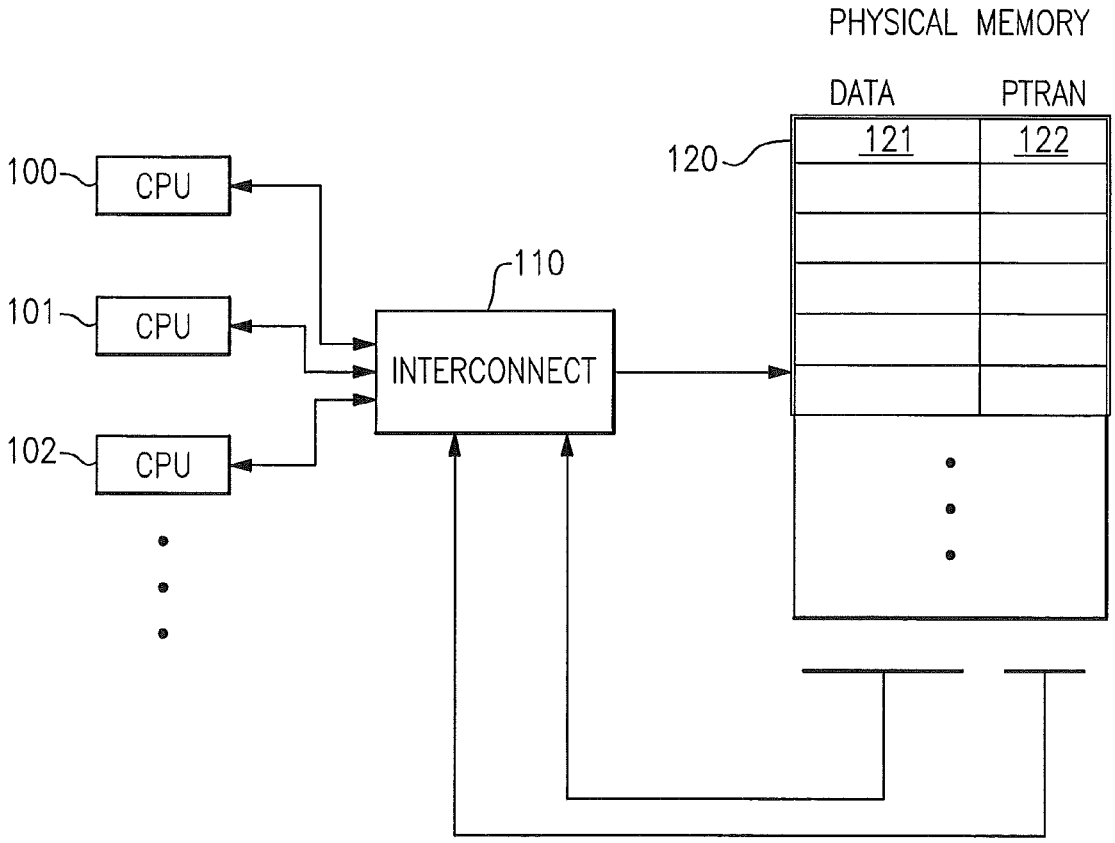

PHYSICAL MEMORY
DATA
PTRAN
121
122
120
100
CPU
101
CPU
102
CPU
110
INTERCONNECT

FIG.1

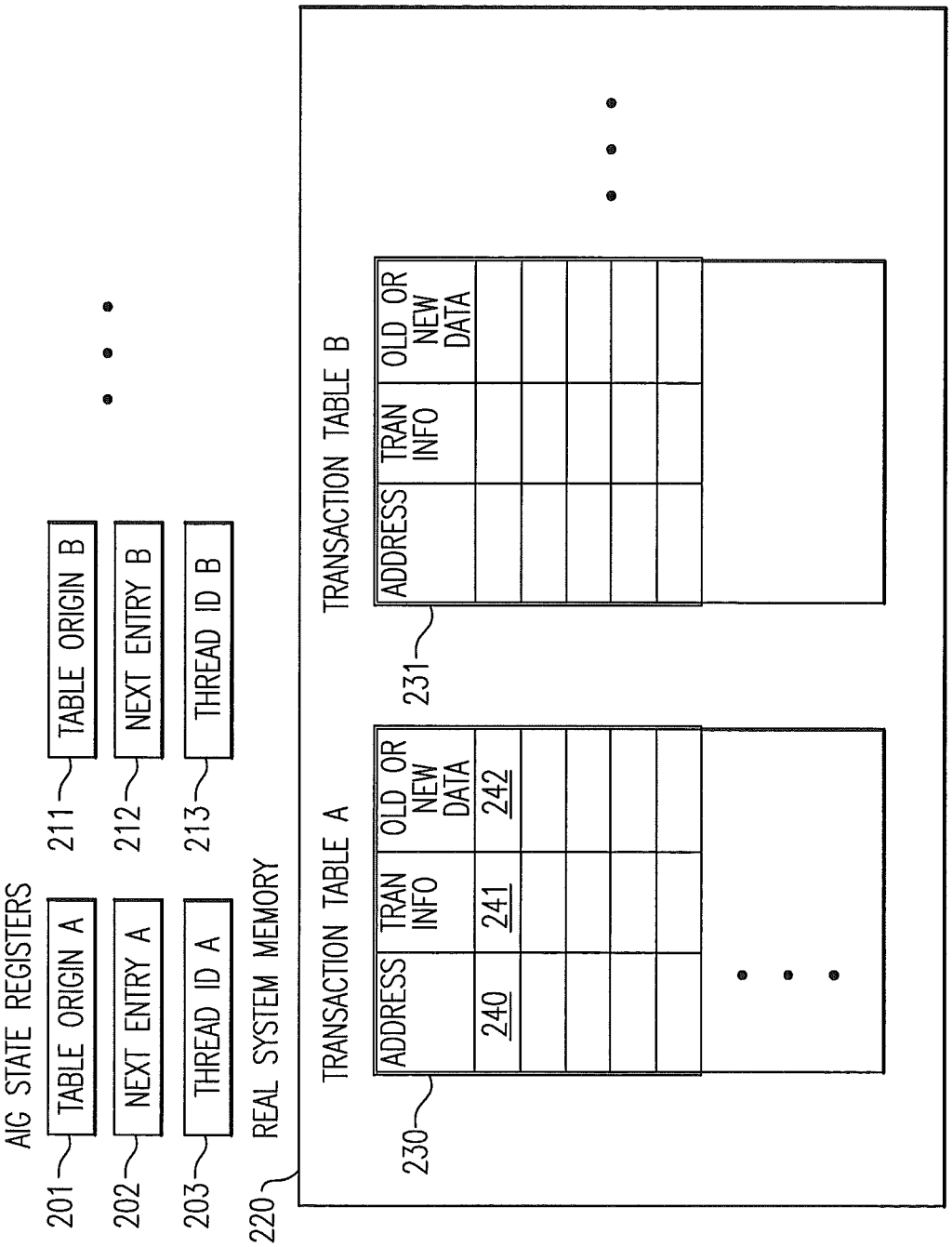

AIG STATE REGISTERS
201
TABLE ORIGIN A
202
NEXT ENTRY A
203
THREAD ID A
211
TABLE ORIGIN B
212
NEXT ENTRY B
213
THREAD ID B
220
REAL SYSTEM MEMORY
230
TRANSACTION TABLE A
ADDRESS
TRAN INFO
OLD OR NEW DATA
240
241
242
231
TRANSACTION TABLE B
ADDRESS
TRAN INFO
OLD OR NEW DATA

FIG.2

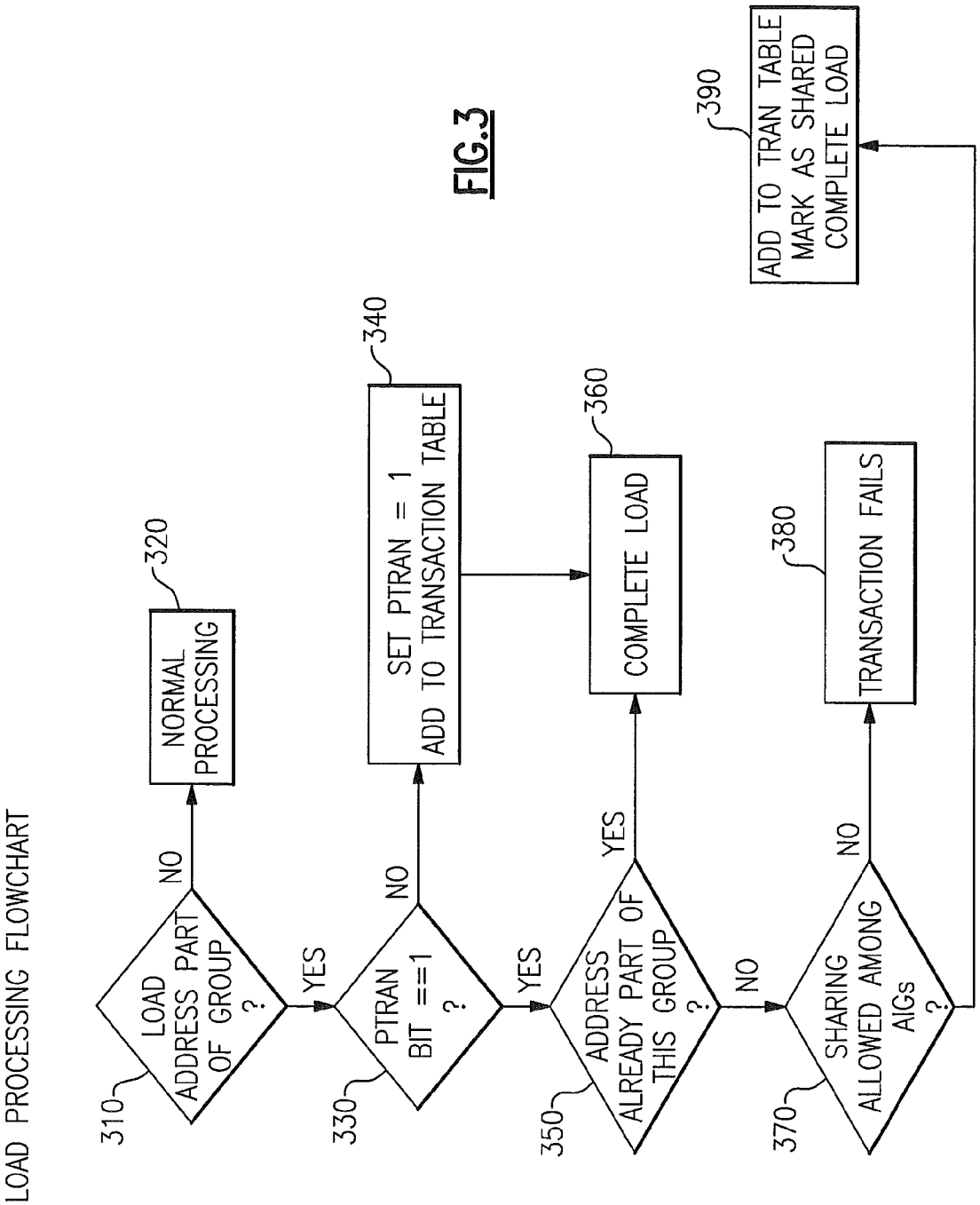

LOAD PROCESSING FLOWCHART
LOAD ADDRESS PART OF GROUP ?
310
NO
NORMAL PROCESSING
320
YES
PTRAN BIT ==1 ?
330
NO
SET PTRAN = 1 ADD TO TRANSACTION TABLE
340
YES
ADDRESS ALREADY PART OF THIS GROUP ?
350
YES
COMPLETE LOAD
360
NO
SHARING ALLOWED AMONG AIGs ?
370
NO
TRANSACTION FAILS
380
ADD TO TRAN TABLE MARK AS SHARED COMPLETE LOAD
390

FIG.3

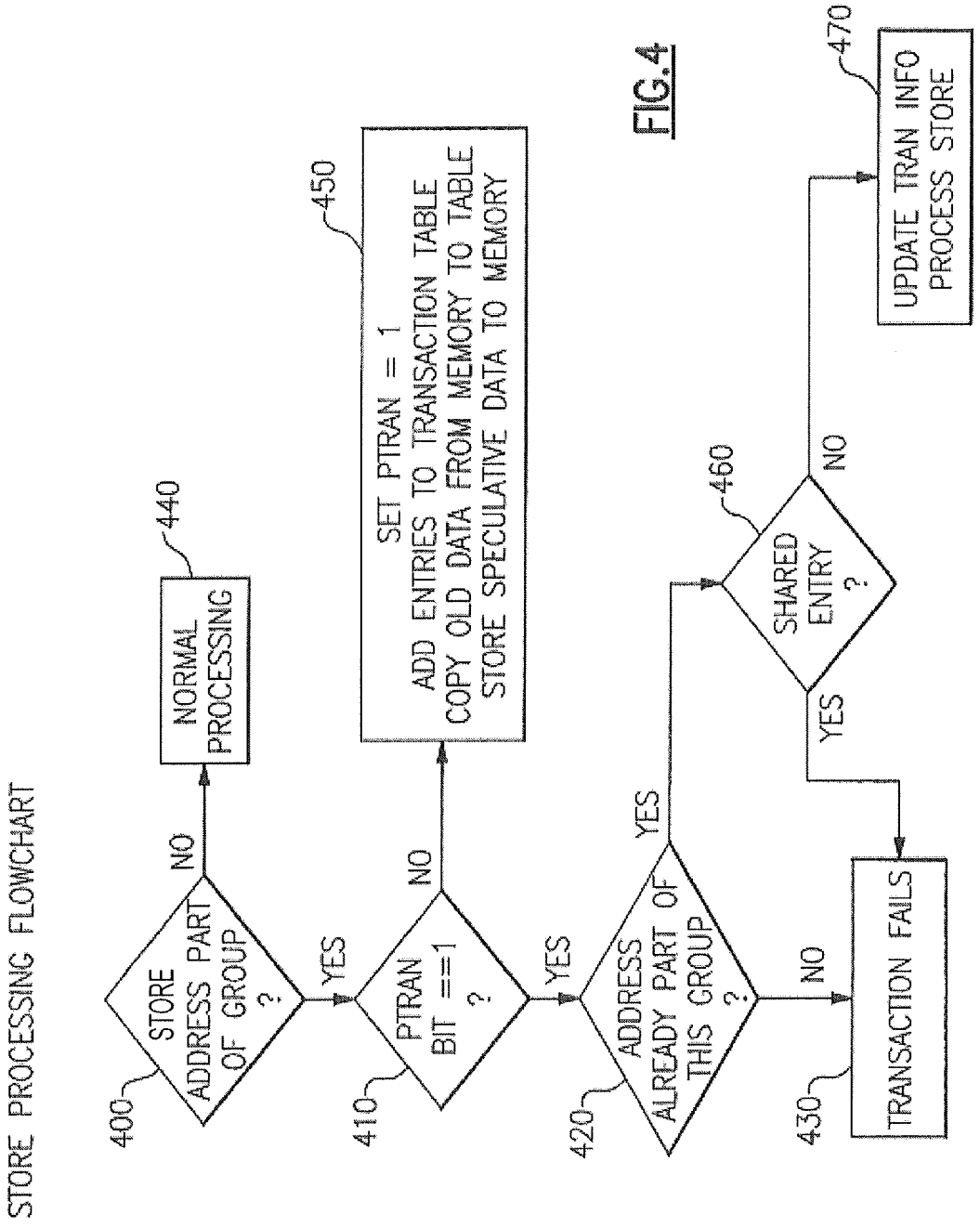

STORE PROCESSING FLOWCHART
400
STORE ADDRESS PART OF GROUP ?
NO
YES
440
NORMAL PROCESSING
410
PTRAN BIT ==1 ?
NO
YES
450
SET PTRAN = 1
ADD ENTRIES TO TRANSACTION TABLE
COPY OLD DATA FROM MEMORY TO TABLE
STORE SPECULATIVE DATA TO MEMORY
420
ADDRESS ALREADY PART OF THIS GROUP ?
YES
NO
460
SHARED ENTRY ?
NO
YES
430
TRANSACTION FAILS
470
UPDATE TRAN INFO
PROCESS STORE

FIG.4

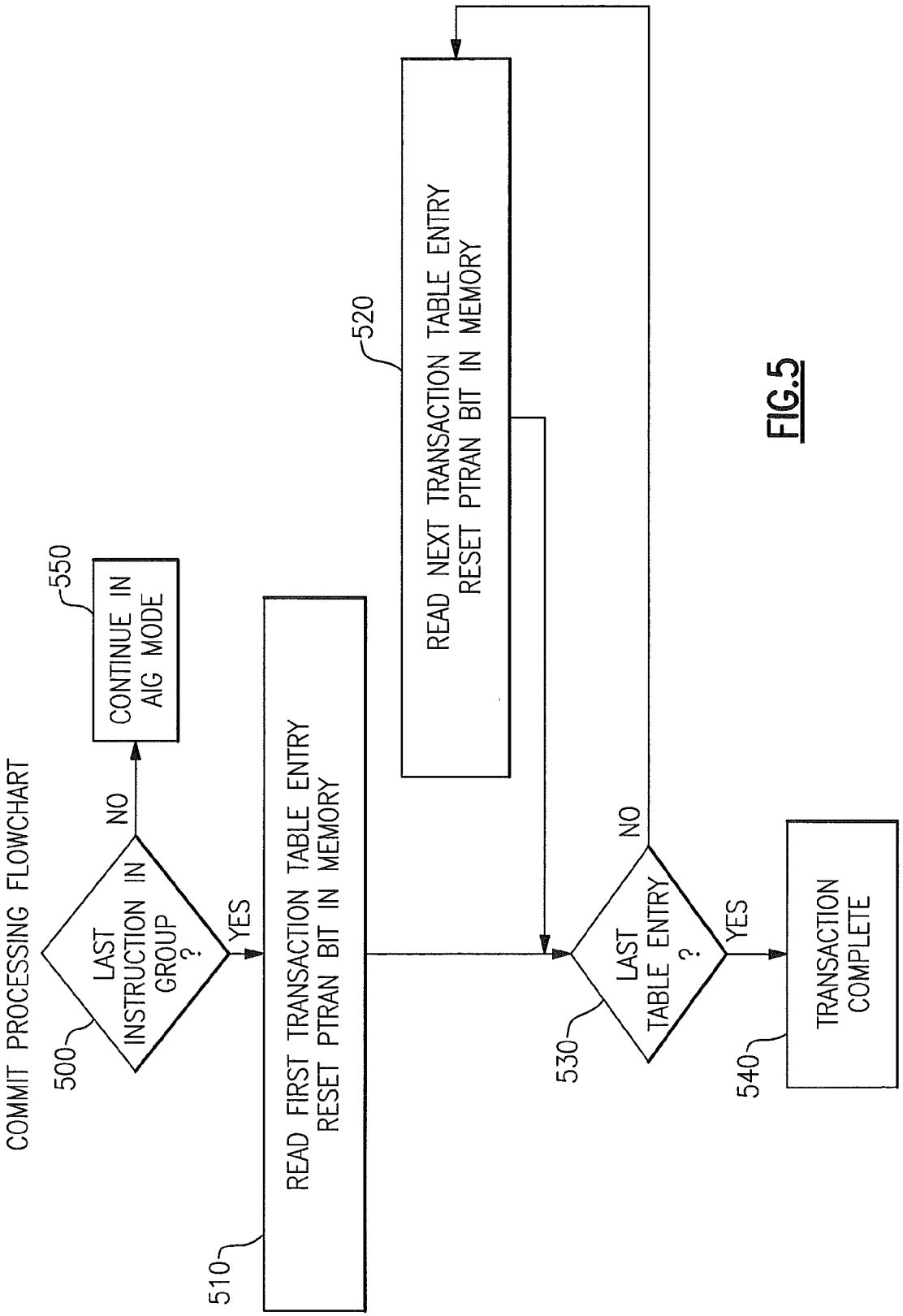

COMMIT PROCESSING FLOWCHART
LAST INSTRUCTION IN GROUP ?
500
NO
YES
CONTINUE IN AIG MODE
550
READ FIRST TRANSACTION TABLE ENTRY
RESET PTRAN BIT IN MEMORY
510
READ NEXT TRANSACTION TABLE ENTRY
RESET PTRAN BIT IN MEMORY
520
LAST TABLE ENTRY ?
530
NO
YES
TRANSACTION COMPLETE
540

FIG.5

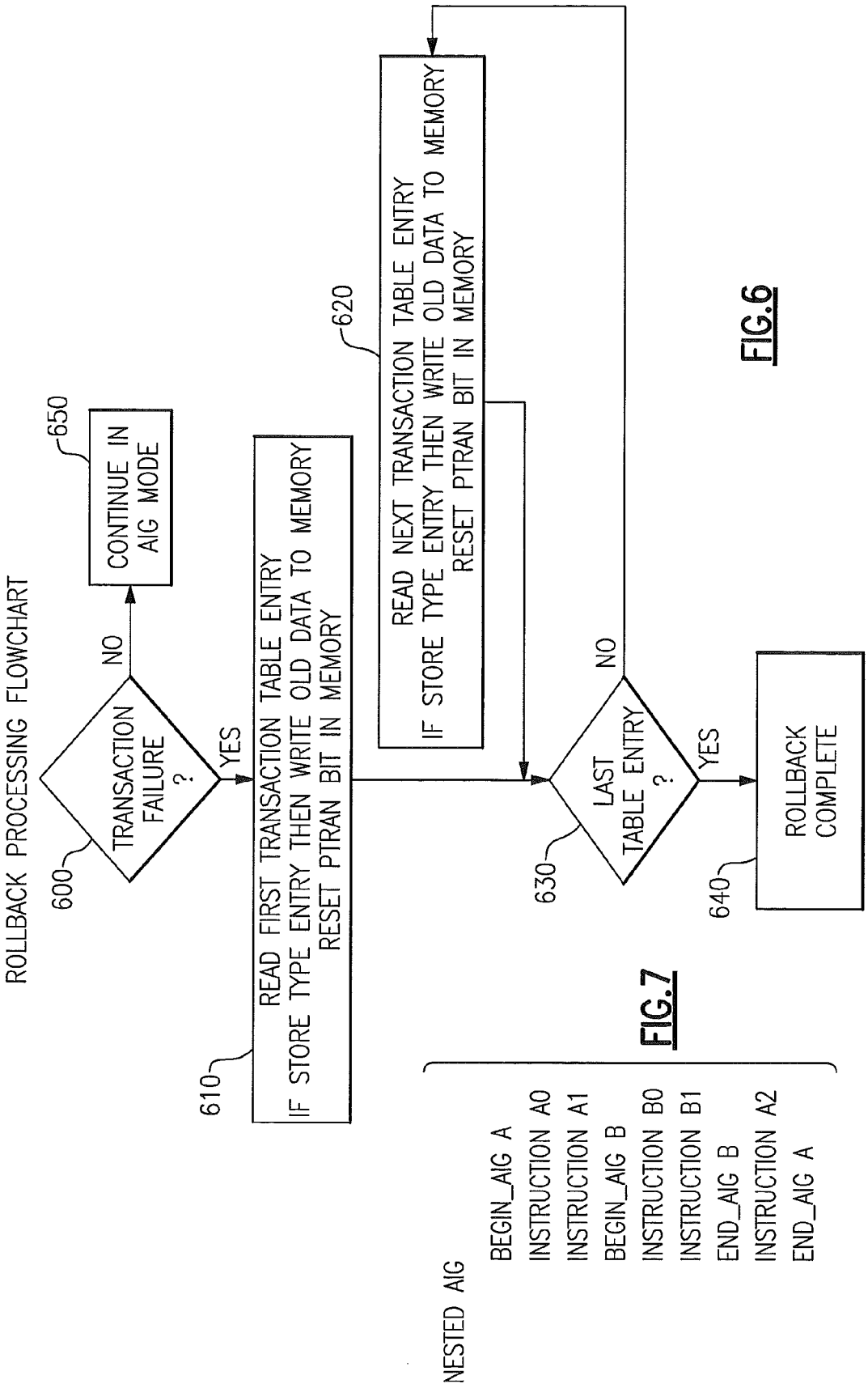


FIG.6

NESTED AIG

BEGIN_AIG A
INSTRUCTION A0
INSTRUCTION A1
BEGIN_AIG B
INSTRUCTION B0
INSTRUCTION B1
END_AIG B
INSTRUCTION A2
END_AIG A

FIG.7

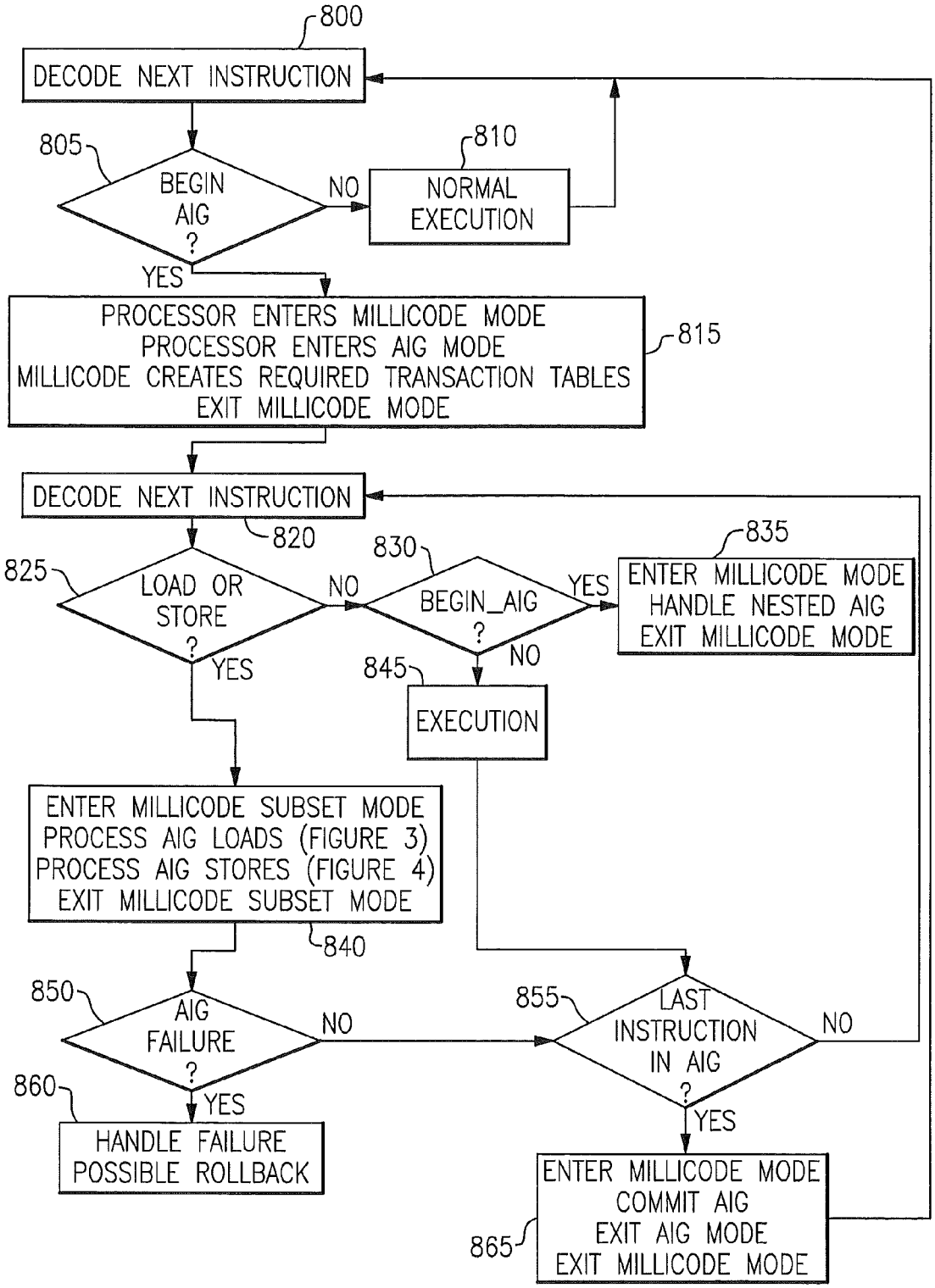

800
DECODE NEXT INSTRUCTION
805
BEGIN AIG ?
NO
810
NORMAL EXECUTION
YES
PROCESSOR ENTERS MILLICODE MODE
PROCESSOR ENTERS AIG MODE
MILLICODE CREATES REQUIRED TRANSACTION TABLES
EXIT MILLICODE MODE
815
DECODE NEXT INSTRUCTION
820
825
LOAD OR STORE ?
NO
830
BEGIN_AIG ?
YES
835
ENTER MILLICODE MODE
HANDLE NESTED AIG
EXIT MILLICODE MODE
YES
NO
845
EXECUTION
ENTER MILLICODE SUBSET MODE
PROCESS AIG LOADS (FIGURE 3)
PROCESS AIG STORES (FIGURE 4)
EXIT MILLICODE SUBSET MODE
840
850
AIG FAILURE ?
NO
855
LAST INSTRUCTION IN AIG ?
NO
860
YES
HANDLE FAILURE
POSSIBLE ROLLBACK
YES
ENTER MILLICODE MODE
COMMIT AIG
EXIT AIG MODE
EXIT MILLICODE MODE
865

FIG.8

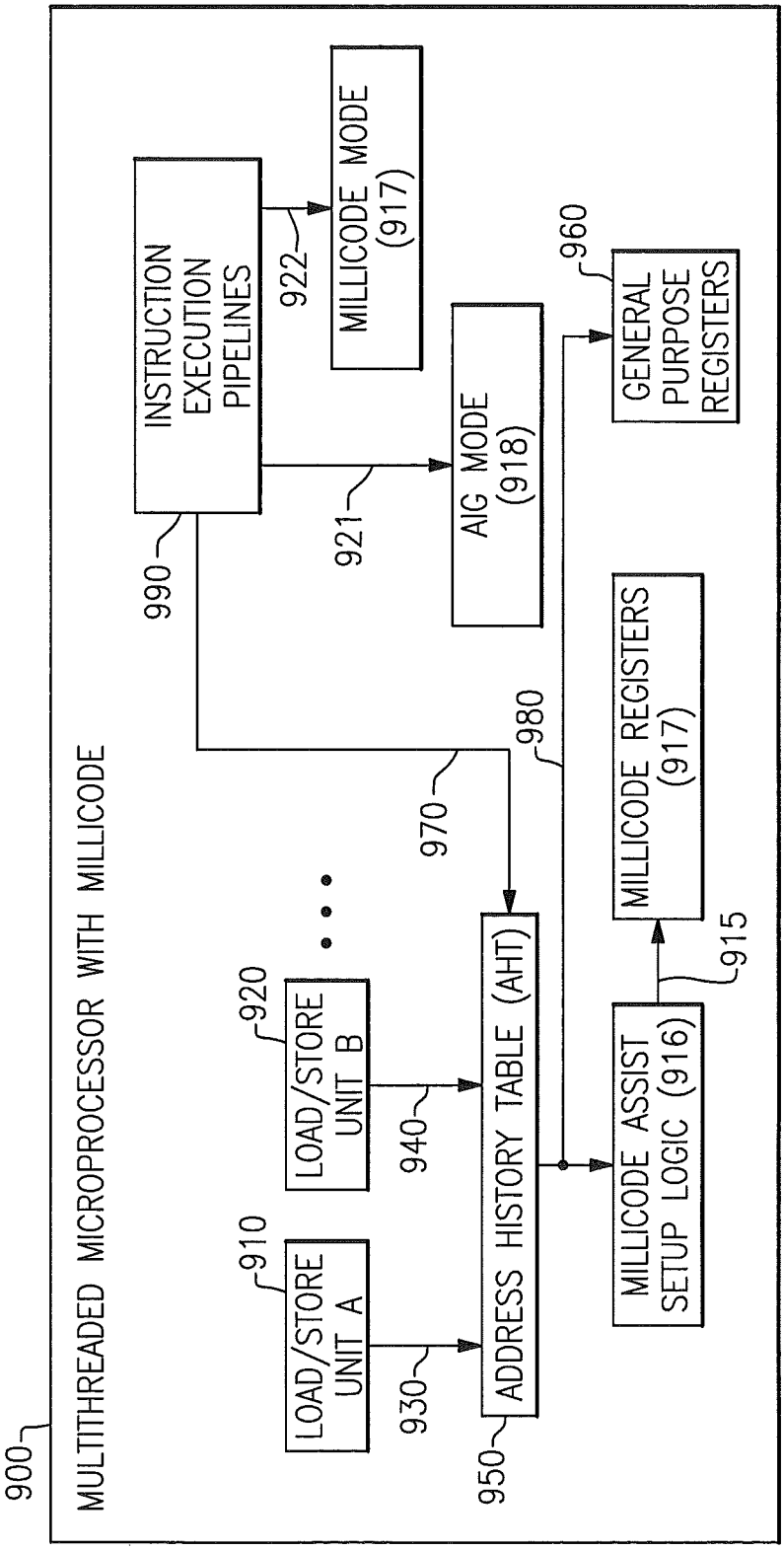

MULTITHREADED MICROPROCESSOR WITH MILLICODE
900
INSTRUCTION EXECUTION PIPELINES
990
922
MILLICODE MODE (917)
921
AIG MODE (918)
960
GENERAL PURPOSE REGISTERS
970
980
LOAD/STORE UNIT A
910
LOAD/STORE UNIT B
920
930
940
ADDRESS HISTORY TABLE (AHT)
950
MILLICODE ASSIST SETUP LOGIC (916)
MILLICODE REGISTERS (917)
915

FIG.9

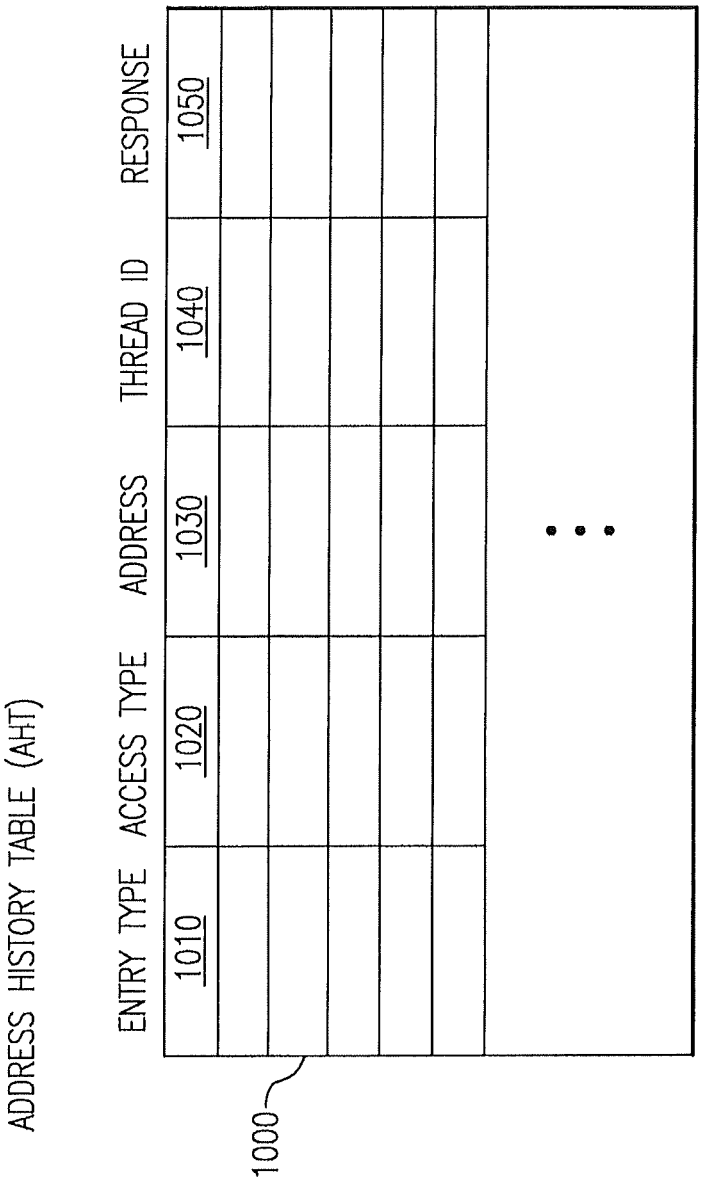

ADDRESS HISTORY TABLE (AHT)
ENTRY TYPE
1010
ACCESS TYPE
1020
ADDRESS
1030
THREAD ID
1040
RESPONSE
1050
1000

FIG.10

- PSEUDO CODE FOR ASSIST THREAD FUNCTION (NO CONFLICT FAST PATH)
  - AIG LOADS CAUSE 1 ADDITIONAL LOAD AND 2 STORES (ONE STORE WILL MISS L1)
  - AIG STORES CAUSE 2 ADDITIONAL LOADS AND STORES (ONE STORE WILL MISS L1)

```
        MOVE SPR->R4                          //PULL NEST ENTRY FROM HARDWARE BUFFER IN LD/ST UNIT
        IF ENTRY !=VALID THEN EXIT            //ONLY WORK ON ENTRIES THAT HAVE L2 RESPONSES
        IF L2 RWESP !=0 THEN BR TO CONFLICT   //IF L2 RESPONSE IS NOT OK THEN RESOLVE CONFLICT
                                              //              ON NEXT PAGE
NORMAL: R5  R4 & MASK                         //OTHERWISE CALCULATE ADDRESS FOR THE
        LOAD (R5) UINTO R8                    //    PART OF THE SUMMARY TABLE TO BE UPDATED
        SHIFT R8
        WRITE THE CORRECT BIT IN R8           //MARK SUMMARY TABLE FOR THIS ENTRY
        SHIFT R8 BACK                         //BOTH LOADS AND STORES NEED TO DO THIS
        STORE R8 BACK TO (R5)

        IF (CMD=LOAD) BR TO DETAIL ENTRY      //LOADS DON'T SET THE STORE SUMMARY
        LOAD (R6) INTO R9                     //SET STORE SUMMARY BIT
        SHIFT R9
        WRITE THE CORRECT BIT IN R9
        SHIFT BACK
        STORE R9 INTO (R6)

DETAIL ENTRY: R10=ADDRESS TO WRITE DETAILS    //GENERATE DETAILED ENTRY
        GENERATE DETAILED ENTRY IN R4
        STORE R4 INTO (R10)
```

FIG.11

COMPUTING SYSTEM WITH TRANSACTIONAL MEMORY USING MILLICODE ASSISTS

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a divisional of U.S. patent application Ser. No. 11/928,533, filed Oct. 30, 2007, which is a continuation in part of U.S. Ser. No. 11/748,044, filed May 14, 2007, the contents of both are incorporated by reference herein in their entirety.

BACKGROUND

This invention relates to computer systems, and particularly to a computing system with transactional memory using millicode assists.

Current multiprocessor and multithreaded computing systems allow the performance of a single software application to be scaled to many times the possible performance of a single threaded application. Current software and hardware systems provide for the parallel processing of multiple threads of execution. Software applications can use existing thread libraries, such as the POSIX pthread library, to control the creation of multiple threads of parallel execution. The use of multiple threads works well for applications that operate on easily partitioned tasks and data. Course grain locks can be used to control access to the few shared data structures to prevent rare conflicts between the data updates of multiple threads.

Many software applications contain data structures that must be shared among multiple threads and have frequent concurrent inspections and updates of the shared data structures. These applications require additional modifications in order to obtain good scaling when using large numbers of threads. Applications which use multiple threads of execution that access shared data structures currently require the use of specialized data locking routines in order to produce a reliable outcome that is free from deadlocks and corrupted data. The majority of existing multithreaded applications in this category use fine grained software locks to achieve good performance and correct operation. Writing high performance multithreaded programs which use fine grained software locks is extremely difficult and requires expert programming skills. The lack of these skills in the software industry severely limits the production of multithreaded applications which require the use of shared data structures and therefore the usefulness of multithreaded and multiprocessor computing systems for certain application classes, including many forms of transaction processing.

Various "Transactional Memory" systems have been proposed and built to provide a simpler programming model for constructing multithreaded applications that need to control access to shared data structures. These systems allow software running on one thread of execution to optimistically assume that shared data structures can be updated without conflict with the accesses and updates of other threads of execution. The speculative updates to memory are kept "pending" until the transactional memory system confirms that no conflicts with storage accesses of other threads have occurred. The transactional memory system must be able to discard the pending speculative updates when conflicts between the storage accesses of multiple threads are detected. The existing transactional memory systems range from those that rely completely on new software constructs to those that rely on a mixture of hardware and software to obtain reasonable performance. Some existing systems have very high overhead in terms of the number of instructions executed in support of the required transactional memory behavior. Other existing systems have limitations associated with complex data cache structures that hold the "pending" updates to memory in caches. The cache based systems use many additional cache coherency states which causes a large increase in the design and simulation efforts for those systems.

Recently transactional memory systems have been proposed as illustrated by Microsoft's U.S. Patent Application Publication No. U.S.2007/0028056 which use software "enlistment records" associated with memory locations that have been accessed by transactions. The Microsoft system also uses a "version value" in the "enlistment record". FIG. **3** of Pub. No. 2007/0028056 includes an operation "Locate enlistment record of memory location". The text describing FIG. **3** gives the example of the enlistment record being part of a software object at the referenced memory address. From Pub. No. 2007/0028056 in paragraph 24, it will be seen that Timothy L. Harris, the Microsoft inventor, indicates that an enlistment record is created with the use of a "software transactional memory interface". The Microsoft Pub. No. 2007/0028056 uses version numbers associated with each enlistment record and needs to deal with cases where the version number exceeds the maximum number supported by the enlistment record. In hindsight, after learning of the details of our invention, it will be recognized that these features of the recent developments in transaction memory systems are not needed and can be improved upon.

Earlier, and now long ago, as pointed out in the development of fast paced computing systems developments summarized by David A. Wood, University of Wisconsin, Transactional Memory Workshop, April 8, 2005, it was Chang and Mergen of IBM (described in 801 Storage: Architecture and Programming) who proposed using a lock bit associated with each segment of virtual memory. Their system provided an ability to detect concurrent accesses of storage locations by multiple threads but restricted the total number of concurrent threads that could operate on any single virtual memory segment when the transaction locking mechanism becomes active. Large virtual memory page tables which are required by Chang and Mergen to execute concurrent threads imposes a performance penalty on all threads of execution that use virtual memory, not just those that execute "transactions, so the Chang and Mergen suggestions became an anecdote in the prior art, as Woods said "No one seems to be looking at what they learned". The current invention uses a "Transaction Table" that is distinct from the virtual memory page tables of the system.

Unknown to Woods, in the Chang and Mergen IBM Yorktown facility there have been ongoing laboratory developments relating to transactional memory systems, culminating with the most recent transactional memory system described by Xiaowei Shen U.S. patent application Ser. No. 11/156913, filed Jun. 20, 2005, and entitled "Architecture Support of Best-Effort Atomic Transactions for Multiprocessor Systems".

Shen describes a transactional memory system which focuses on using "caches as buffers for data accessed by atomic transactions". Xiaowei Shen forces the failure of a transaction when the system detects a "buffer overflow" and does not propose dealing with the case of speculative data being evicted from the cache. A buffer overflow will result if too many transactional loads or stores target the same cache congruence class. The percentage of transactions which over-

flow the cache and fail will be proportional to the number of loads and stores contained in a transaction. Long transactions will fail more often.

Other systems have been proposed which use a cache to implement a conflict detection scheme, and to capture speculative data which overflows the cache. We have found it desirable not to rely primarily on data cache states for conflict detection. The required specialized cache states cause an undesired increase in complexity and also makes it difficult to add the required detection and isolation mechanisms to existing multiprocessor cache coherency designs. Nevertheless there are numerous attempts to exploit the cache which have been tried, besides the Xaiowei Shen development, including, Moore et al who describe the use of a "before-image log" in their paper "LogTM: Log-based Transactional Memory". Moore uses cache coherency states to implement a conflict detection scheme. Another such system, Ananian et al describe a transactional memory system which uses a single "unsorted linear array data structure" to capture speculative data which overflows data caches. They use an extra bit per cache set, the "O" bit to indicate if that set has "overflowed", and another bit per cache entry, the "T" bit to indicate that the entry holds speculative data. The linear array data structure does provide the ability to support longer transactions but it does not provide the fast detection capability of the current invention. The linear array in Ananian needs to be searched for any cache access that targets the set that "overflowed". This can be a long process for transactions which cause many overflows. Rajwar, Herlihy and Lai take a similar approach as Ananian in their paper "Virtualizing Transactional Memory". Rajwar also uses a data cache as the primary mechanism to track the speculative state associated with transactions. Speculative data which is forced out of the caches is moved to the XADT overflow area in virtual memory. The detection of a possible conflict with an address that is part of the XADT requires a slow linear search of the XADT. Rajwar describes the use of filters to eliminate some of these searches but there are many cases where the searches will still need to be done.

In addition to the above summarized developments in the field, many software interfaces and instruction set modifications have been proposed for the support of transactional memory. The paper "Architectural Semantics for Practical Transactional Memory" (McDonald et al, Computer Systems Laboratory, Stanford University, 2006—this paper listed in our IDS is submitted herewith and is incorporated herein by reference) compares some of the proposals and provides references for many others and is incorporated herein by reference.

SUMMARY

The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a computing system with transactional memory using millicode assists. The millicoded microprocessors have millicode that is invoked for certain cases. Memory accesses generated by instructions within an Atomic Instruction Group (AIG) transaction are held pending. A new instruction (e.g. "tend") or a buffer full indication causes the millicode to invoked to perform load and store processing. The millicode providing transactional memory functions including creating and updating transaction tables, committing transactions and controlling the rollback of transactions which fail. Millicode Assist Setup Logic provides the ability to move the contents of the Address History Table to positions in millicode registers that are well suited to fast execution using the native hardware instructions of the microprocessor.

The program millicode provides assisting thread functions including AIG mode loads causing one additional load and two stores in which one store will miss the L1 cache and AIG mode stores which causes two additional loads and three stores in which one store will miss the L1 cache.

The use of hardware assists utilizing Millicode Assist Setup Logic for simple functions and millicode for more complex functions allows the invention to provide a balance of speed and flexibility when implementing the complex operations required for the support of a transactional memory programming environment in many types of microprocessors.

Transactional memory uses a combination of a "private to transaction" (PTRAN) tag, attached to each increment of real system memory, and a log of speculative loads and stores to provide an improved implementation of a transactional memory system. The current invention uses a log with the added mark bit employed as a "private to transaction" (PTRAN) tag and associated with every increment of real system memory. Hardware is provided to quickly detect conflicts between the storage accesses of transactions running on multiple threads of execution. The use of the tag in memory and associated conflict detection hardware included in this invention provides a much faster transactional memory system with much less overhead when compared to existing systems. The complexity of the current invention is lower than prior attempts at using additional cache coherency states for conflict detection, especially for systems with large numbers of processors and associated interconnections.

The current invention uses the main memory array of the computing system to hold the speculative data and can support very long transactions. The current invention can benefit from the use of data caches but it does not require their use. The current invention provides the ability to check the address of a new memory access without a long search process for common cases. Likewise, it is an improvement over software only transactional memory systems since the conflict detection and use of the tag in memory eliminates some of the software overhead associated with tracking the speculative state of transactions. It also provides the ability to detect storage conflicts at a very fine level (own to a single byte) as opposed to prior art software systems that track updates to entire software objects which may be hundreds or thousands of bytes. Existing software systems will either give frequent over-indication of potential data conflicts or incur very large software path length penalties when attempting to track the updates to individual components of software objects.

Many software interfaces and instruction set modifications have been proposed for the support of transactional memory. The current invention can be used in combination with any of them in order to provide high performance transactional memory operations without incurring a large increase in hardware or software complexity. The preferred embodiment is described for the PowerPC architecture but anyone skilled in the art could apply the same approach to any other architecture such as IBM's zSeries, IBM's pSeries with the P3, P4, P5 processors, and even IBM's System 38 and its AS/400 which have a memory work of 65 bits could utilize the support of this invention, as well as other computer systems, such as Sun Microsystems' SPARC, Intel's IA32 etc. Anyone skilled in the art could extend the current invention for use with other Application Programming Interfaces (APIs) that may be created for other specialized versions of transactional memory implementations.

The current invention uses a hardware bit or bits associated with all memory locations, not just those that are currently part of an active transaction. The current invention uses the

added hardware bit or bits to provide much faster execution of transactions than that which can be obtained using Microsoft's Pub. No. 2007/0028056. The current invention uses an improved transaction table as a log to optimize the memory usage and provides a system which uses less memory than would be used by other systems like the Microsoft proposal, yet the invention can execute applications developed for the Microsoft proposal. The current invention provides one or more PTRAN hits for every storage increment. There is no need in the current invention for an indirect method of "locating" the PTRAN bit. The current invention will provide a much faster indication of potential conflict since the PTRAN bit is a directly addressable part of the memory location to be marked. The current invention does not need a Microsoft style interface, and achieves its benefit using hardware and firmware to update and reset the PTRAN bit. The current invention does not require application software to be aware of the state of the PTRAN bit or bits although one may provide a direct interface if desired. The current invention is also optimized for a short commit processing time. The current invention does not require the use of a version number for each storage location.

The current invention uses a hardware bit or bits associated with all memory locations, not just those that are currently part of an active transaction. The current invention uses the added hardware bit or bits to provide much faster execution of transactions than that which can be obtained using Microsoft's Pub. No. 2007/0028056. The current invention uses an improved transaction log method to optimize the memory usage and provides a system which uses less memory than would be used by other systems like the Microsoft proposal. The current invention provides one or more PTRAN bits for every storage increment. There is no need in the current invention for an indirect method of "locating" the PTRAN bit. The current invention will provide a much faster indication of potential conflict since the PTRAN bit is a directly addressable part of the memory location to be marked. The current invention does not need a Microsoft style interface, and achieves its benefit using hardware and firmware to update and reset the PTRAN bit. The current invention does not require software to be aware of the state of the PTRAN bit or bits although one may provide a direct interface if desired. The current invention is also optimized for a short commit processing time. The current invention does not require the use of a version number for each storage location.

We noted above the Chang and Mergen suggestions which became an anecdote in the prior art, as Woods said "No one seems to be looking at what they learned". An advantage of the current invention is that it uses a "Transaction Table" that is distinct from the virtual memory page tables of the system. This enables the current invention to process very large transactions and to provide conflict detection down to a single byte. The current invention has no limit on the number of concurrent threads that can simultaneously access memory locations in the same virtual memory segment. The current invention performs conflict detection on physical memory addresses (or real addresses) and only restricts simultaneous access at the granularity of this conflict detection. This granularity can differ for various implementations of the current invention but will usually be on the order of bytes. If the Chang and Mergen system tries to use large numbers of concurrent threads, that system could not provide similar benefits without using extremely large virtual page table formats and consequently incurring a performance penalty. The current invention does not impose this performance penalty.

System and computer program products for implementing transactions using the above-summarized methods are also described and claimed herein. Generally computer program products are delivered as computer program media which are tangible embodiments of the program delivering the instructions via a CD Rom, a computer disc drive, a connection to another system or other tangible embodiment of a signal for delivering a program signal to the computing system which supplies the support of this system, all collectively known as computer program media. The computer program media may be provided as an independent software medium installed in the computing system memory or installed as firmware embodied in the computing system memory itself during operation.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention. For a better understanding of the invention with advantages and features, refer to the description and to the drawings.

As a result of the summarized invention, technically we have achieved a solution which enables any millicoded processor to be used for transactional memory systems and provides a much faster transactional memory system with much less overhead when compared to existing systems. The current invention is also an improvement over existing hardware based transactional memory systems that rely on changes to cache coherence protocols. It allows the hardware system to deal with transactions which are long enough to overflow average size caches and doesn't involve the virtual memory management overhead of prior art schemes. The current invention has a much lower level of hardware complexity and is easier to implement and verify via simulation. It also allows for the ability to detect conflicts at a finer granularity than the cache line granularity of prior art systems that are tightly coupled with data caches. The current invention also allows for the fast execution of nested transactions.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

FIG. **1** illustrates our computing system with optimized transactional memory using physical memory.

FIG. **2** illustrates our real system memory transaction tables.

FIG. **3** shows our load processing flowchart.

FIG. **4** shows our store processing flowchart.

FIG. **5** shows our commit processing flowchart.

FIG. **6** shows our rollback processing flowchart.

FIG. **7** illustrates new special instructions and Atomic Instruction Group (AIG) of instructions used in a nested Atomic Instruction Group (AIG).

FIG. **8** shows the flow of decoding and execution of instructions in a computing system that uses the current invention.

FIG. **9** shows how the current invention uses an Address History Table (**950**) that is part of a multithreaded microprocessor (**900**) that has been designed for support of transactional memory and which uses millicode for the execution of complex instructions.

FIG. **10** illustrates more detail of the Address History Table (**1000**).

FIG. **11** illustrates in pseudo code the code for assist thread functions.

DETAILED DESCRIPTION

The detailed description explains the preferred embodiments of the invention, together with advantages and features, by way of example with reference to the drawings.

Turning now to the drawings in greater detail, it will be seen that in FIG. **1** there is a computing system illustrating an embodiment of our invention which has one or more microprocessors (**100**,**101**,**102**) coupled to a physical memory array (**120**) via an interconnection element (**110**). The physical memory array stores transaction data (**121**) and "private to transaction" (PTRAN) tags (**122**) which are associated with every increment of real system memory. The interconnection element (**110**) can be implemented as a shared bus or crossbar switch. The invention applies to systems which use any other scheme of interconnecting physical memory to a multiprocessor system which may be implemented in one or more chips. The memory could be broken down into smaller portions and distributed across private connections to each of the CPU chips as done for the IBM Systems using the Power4 microprocessor or for the AMD Opteron based servers. The microprocessors and memory controllers may be located together on a single silicon chip or they may be spread across multiple chips.

The physical memory of the computing system is divided into n increments. One or more "private to transaction" bits (PTRAN) associated with every increment of real system memory are provided for each of the n increments. The invention allows for the choice of any memory increment size and the best choice will depend on workload characteristics, hardware costs and data caching structure used in the target system. An increment of 16 bytes is used in the illustrated embodiment. The PTRAN bit(s) are used to indicate whether (or not) a data entry in memory is part of the speculative memory state of an uncommitted transaction that is currently active in the system.

Special new instructions (BEGIN_AIG, END_AIG) as illustrated by the Nested AIG code sequence of FIG. **7** are used to mark the beginning and end of a group of instructions. The instructions which execute between the special new instructions are referred to as an "Atomic Instruction Group" (AIG) illustrated by AIG instructions shown in FIG. **7** (Instruction A**0**, Instruction A**1**, Instruction A**2**). Additional storage access rules are used when a processor is executing instructions which are part of Atomic Instruction Group. All of the storage locations modified by the AIG group of instructions are updated in memory in an atomic fashion. The updates to the storage locations are kept "pending" until the processor and/or software application indicates that they should be "committed". All of the updates are either committed to "normal" memory at once or they are discarded. The results are discarded when hardware and/or software detects a conflict between the storage accesses of multiple AIGs that are executing concurrently in the multiprocessor system. The invention provides a way for hardware to quickly detect potential conflicts between the storage accesses of multiple AIGs. Although the invention uses special new instructions to mark the boundaries of a transaction, any other method could be used to identify a group of memory locations that are to be updated in an atomic fashion. The invention is compatible with any number of software interfaces that may be used to implement a transactional memory system. The invention can provide the same fast conflict detection for any system which is attempting to provide an atomic update of multiple storage locations. The invention also applies to systems which mark the boundaries of an instruction group in any other ways, including compiler generated hints attached to other instructions, internal microprocessor commands generated by internal microcode or millicode.

The PTRAN tag is one or more bits associated with an increment in memory which is set for all memory accesses generated by instructions that are part of an Atomic Instruction Group. A processor inspects the bit before attempting to set it; this enables the quick detection of potential conflicts with other AIGs that are concurrently executing on other threads. The setting of the bit may be accomplished by a TS "test and set" operation of the IBM z/Architecture (as described by the IBM z/Architecture Principles of Operation) or any other equivalent operation that enables an atomic update in a multithreaded or multiprocessor system.

Turning now to FIG. **2**, it will be seen that Transaction Tables (**230**,**231**) are created as part of the real system memory (**220**) which is illustrated here as physical memory. The Transaction Tables could also be created in logical or virtual memory. Any system to map the logical system memory to the physical system memory can be used, and there are numerous examples known in the art which can be used, such as those in the IBM zSeries, IBM's pSeries, Sun Microsystems' SPARC, Intel's IA32 etc. A Transaction Table entry is made when instructions that are part of an Atomic Instruction Group cause a memory location to be inspected or updated. A Thread ID (**203**,**213**) is associated with each Transaction Table. A Next Entry register (**202**,**212**) is used as an index into the Transaction Table and indicates which entry should be written next. A Table Origin (**201**,**211**) indicates the address of the first entry of the Transaction Table for its thread (**203**, **213**). The Transaction Table is used to hold additional information beyond the simple information that is associated with the PTRAN tag bit or bits that are associated with the memory location that has been inspected or updated. This additional information is related to the speculative state associated with an Atomic Instruction Group. In an illustrated IBM zSeries or pSeries CPU (**100**, **101**, **102**) embodiment which we describe here we prefer to use a single PTRAN bit. In this case the Transaction Table will contain all additional information about the speculative state, therefore the PTRAN bit will only indicate that a physical address is involved in an AIG. This is illustrated for Transaction Table A (**230**) as the address (**240**), the transaction info identifier (**241**) and old or new data (**242**). Other embodiments of the current invention may use additional PTRAN bits which can be used by hardware or software to speed the processing of certain events involved in the processing of Atomic Instruction Groups or the speculative states of any other transactional memory system. The combination of using just a single bit in memory and associating that bit with a more complete description of the transactional state in the Transaction Tables provides a transactional memory system which requires very little hardware overhead without incurring the performance penalty seen in prior-art software transactional memory systems.

Turning again to FIG. **2**, it will be seen as we said that an entry in the Transaction Table, A for example (**230**), comprises the address (**240**) that has been inspected or updated inside of an AIG, a Tran Info field (**241**) and a Data Field (**242**). A store instruction that is part of an AIG will cause the system to copy the "old" data value from the original storage location to the Transaction Table entry for that address and the new speculative data is placed in the main storage location.

The invention can be made to work with the new values held in the Transaction Table if desired. The preferred embodiment places the "old" data value in the transaction

table. This allows the system to be optimized for the case where most transactions are successful. The old data can be discarded quickly when it is no longer needed, when a transaction is committed permanently to memory, by changing the pointer to the transaction table or by clearing the contents of the transaction table. The Tran Info field of a Transaction Table entry (**241**) includes any transaction information that is needed to make detailed decisions about the need to cause a transaction failure. It also has provisions for additional information to allow efficient support of nested transactions, virtualized transactions or other extensions of the transactional memory architecture. In the preferred embodiment, the Tran Info field includes an indication of whether the storage access of the associated address (**240**) was a load type access or a store type access. The Tran Info field (**240**) can also indicate whether the address is shared among multiple AIGs in the system.

FIG. **3** shows a load processing flowchart for the actions for tracking a speculative state using the PTRAN bit and the Transaction Tables. When a processor attempts a load type access, initially a decision (**310**) is made to determine whether the load access address is part of an AIG and whether the special storage access rules for AIG accesses apply. If not, normal processing applies and a normal load processing (**320**) operation is followed. Whether or not a load type access is part of an AIG may be determined in many different ways in the decision process step (**310**). A mode bit may be set in the processor pipeline or in load/store units to indicate that a special AIG mode of operation is active. The mode may be part of the physical state of the processor or the logical state of a virtual processor. If the special mode of operation is not active then the load is treated as a normal load (**320**).

An "override" of this "AIG active mode" may be provided. The override could be associated with the logical memory segment or logical memory page which contains the target address. An override forces the system to treat the storage request as normal in spite of the fact that the request is part of an AIG. If the override is active then the result of the decision (**310**) will cause normal processing (**320**) to take place. Assuming that an AIG is active and the override is not, then the associated PTRAN bit is inspected at an inspection step (**330**). A load instruction inside of an AIG detects upon inspection the state of the PTRAN bit. When the PTRAN bit is already set (**330**) it is due possibly to the actions of another thread executing on the same processor or on another processor. If at the inspection step **330** it is found that the PTRAN bit is not set (**340**) then the processor sets the PTRAN bit and make a record of the access in the Transaction Table (**230**) for the active AIG by adding to the Transaction Table at the Set PTRAN step (**340**) and then the complete load can continue (**360**). If the PTRAN bit is already set, the address which caused this potential conflict is compared with the addresses already entered in the processor's Transaction Table. If the PTRAN bit was set when tested at the inspection step (**330**) by another load earlier in the same AIG, it is already part of the AIG as tested and determined (**350**) and then the complete load may continue (**360**). Each Transaction Table (**230**, **231**) contains the addresses for a particular AIG. Accordingly, if the address was not already part of the AIG as tested and determined (**350**) then the address for the load is not found in the processor's Transaction Table and then the processor checks whether the address is enabled for sharing among AIGs in a determination step for AIG sharing (**370**). If the address is not currently enabled for sharing among multiple AIGs the processor may signal other processors in the system to request a "shared AIG access" for this address during the determination step whether sharing is allowed among AIGs (**370**). A shared access can be granted if no other AIG in the system has speculatively written the storage location. If the shared access is not granted then the AIG fails (**380**), otherwise (**390**) the load address is added to the Transaction Table (**230**) and the Tran Info field (**241**) is updated to indicate that the address is a load access that is shared among multiple AIGs in the system.

The FIG. **4** Store Processing Flowchart shows the actions for processing a store that is part of an AIG. When a processor attempts a store type access initially a store decision (**400**) is made to determine whether the access is part of an AIG and whether the special storage access rules for AIG accesses apply. The mode of the processor is checked in a similar method as previously described for load accesses. If the special mode of operation, under which the special access rules for AIG accesses apply, is not active then the store is treated as a normal store (**440**). Assuming that an AIG is active, the associated PTRAN bit is inspected at the store process inspection step (**410**). If the PTRAN bit is not already set then the PTRAN bit is set at the store transaction step (**450**) and a new entry is added to the Transaction Table (**230**). The "old data" is moved to the Transaction Table entry (**242**), the address is written to the new entry (**240**) and the Tran Info field (**241**) is updated. The Tran Info field (**241**) is marked to indicate that the access associated with this entry was a store type access. The new store data is written to memory after the setting of the PTRAN bit is completed. If the inspection of the PTRAN bit (**410**) indicates that the bit was already set then a decision (**420**) is made based on whether the current store address is already part of an AIG which is active on the processor. The Transaction Table (**230**) for the processor is examined, if it is determined that the address is present in the Transaction Table then the Tran Info (**241**) for the entry is checked and a decision as to shared access entry (**460**) is made. If the Tran Info indicates that the entry is a load type access entry that is shared among multiple AIGs, the transaction fails (**430**) otherwise the Tran Info field for the associated entry is updated to indicate a store type access and the store is processed (**470**).

Turning now to the Commit Processing Flowchart of FIG. **5**, it will be seen that the invention includes a set of actions completed when the last instruction in an AIG has been processed and the entire group is ready to be "committed" permanently to memory as determined initially at a test step (**500**). If not, the processing continued in AIG mode (**550**). An AIG is committed to memory when the processing of the loads and stores of the AIG according to flowcharts from FIG. **3** and FIG. **4** does not result in a Transaction Failure. In the case of AIG success then testing (**500**) determines the last instruction in an AIG has been processed and the entire group is ready to be "committed" permanently to memory. Then the Transaction Table is examined (**510**,**520**) and each entry for the AIG to be committed is read and its associated PTRAN bit is reset in memory at the Transaction Table commit step (**530**) determination.

A specialized hardware engine may be used to complete the performance of this commit operation. A combination of processor caches, multiprocessor coherency actions and the current invention can be used to provide software with the illusion that all of the memory updates for a single AIG occur simultaneously even though the main memory storage arrays are not updated simultaneously. During the commit processing, the resetting of the PTRAN bits continues until the last valid entry in the Transaction Table has been determined to be reached (**530**). At this point the AIG is considered to be committed and the performance by the engine therefore completes (**540**).

Some conditions prevent the completion of an AIG. These conditions may be detected during load processing while executing an AIG (**380**) or during store processing while executing an AIG (**430**). There are many other possible processor conditions that may cause the need to abort the processing of an AIG. These include error conditions detected in the system as well as other conditions that would require significant additional hardware support to enable the processor to handle them correctly. Many prior-art transactional memory architectures include provisions for the abort of transactions and for a subsequent retry. Prior-art software constructs can be used together with the current invention to eliminate the need to provide hardware to deal with all possible special cases. A simple example is the case of a timer interrupt in the middle of processing an AIG. The interrupt may cause the processor to spend a large amount of time running code that is not part of the partially completed AIG. It may not be desirable for the processor to keep the AIG active during this time. The system can force a transaction failure for any AIG that is currently executing when a timer interrupt occurs. A similar approach can be used for any other special case events occurring in the processor.

Transaction failures or forced retries are handled according to the process of the Rollback Processing Flowchart for "rollback" processing shown in FIG. **6**. Turning now to FIG. **6**, it will be seen that the lack of a transaction failure condition (**600**) allows the processor to continue in the AIG active mode (**650**). The preferred embodiment of the current invention uses an "eager" policy with respect to detecting transaction failures and causing transaction rollback. Also, the invention may be used in systems that wait until the end of a transaction to take the actions required for a rollback. The memory updates executed as part of an Atomic Instruction Groups are either committed to normal main storage at the same time (FIG. **5**) or they are discarded with a "rollback" operation (**620**-**640**).

Upon finding a transaction failure condition (**600**) failure several additional steps are required (**610**,**620**). Rollback Transaction Table processing entry steps provide that the Transaction Table (**230**) for the AIG is inspected and any "old" data (**242**) is written back to the main memory address (**240**) indicated in the entry. The PTRAN bit for the associated address is reset. The rollback processing (**620**) continues until the last valid entry in the table has been processed (**630**). After the last valid entry has been processed, the rollback is complete (**640**). The actions taken by the processor at this point will differ based upon various software architectures for transactional memory. Any of transactional memory architectures described in the background may be used. In some cases the AIG will be retried from the beginning. In other cases special software handlers will be invoked to deal with the transaction failure. The current invention may be used with any of these different architectures.

The current invention supports the execution of "nested" transactions. A second AIG may be included within the scope of the first AIG as shown in FIG. **7**. Each BEGIN_AIG special instruction (FIG. **7**) statement causes the system to create a Transaction Table (**230**) and to associate the table with the AIG. Decisions about whether or not a storage address belongs to an AIG (**350**) may include the inner AIG (A) or the combination of the inner and outer AIGs (A & B). The END-AIG special instruction statement (FIG. **7**) ends the transaction sequence, but as shown, a sequence for a specific transaction (Transaction B) may be nested within another sequence (Transaction A) The use of multiple Transaction Tables (**230**) may be used to support many nesting architectures for transactional memory. Multiple versions of the "old data" may be stored in any number of Transaction Tables (**230**) at any nesting depth. Prior-art transactional memory systems that rely on data caches to hold speculative state are unable to provide similar support for nested transactions without adding additional state information to cache directories and adding additional complexity to the cache coherency protocol. Nesting support on prior-art systems would impose a large hardware complexity penalty on those designs. The current invention can also be extended to include a combined Transaction Table that includes entries from both the inner AIG (Instruction B**0**, Instruction B**1**) and the outer AIG (Instruction A**0**, Instruction A**1**, Instruction A**2**). This Combined Transaction Table may be used in place of the individual Transaction Tables or in addition to the individual Transaction Tables.

There are many additional hardware features that can be added to the invention to speed the processing of the Transaction Table manipulations and the setting and resetting of the PTRAN bit(s). Since the PTRAN bit is part of the main storage data, it can be cached in the normal data caches of the system. The Transaction Tables are also part of main storage and can also be cached. Additional control information can be added to the data caches to indicate whether a specific address has been enabled for "shared AIG access" and therefore eliminate the need to search the Transaction Table for some cases.

FIG. **8** shows the flow of decoding and execution of instructions in a computing system that uses the current invention. The preferred embodiment of the invention uses a BEGIN_AIG instruction to indicate the beginning of an AIG and an END_AIG instruction to indicate the end of an AIG. BEGIN_AIG and END_AIG are marking code in the form of instructions representing any chosen equivalent function instruction or other suitable method of identification of a function marks the beginning and end of a group of instructions. These alternatives for marking code can be used. The instructions or marks may be inserted in the code explicitly by programmers in a high level language or added by compilers or translators as implementations of high level locking functions. The instructions or marks may be added by a library function call or be included by a special event in a runtime environment. The instructions or marks may be generated by firmware, hardware or a combination of both as a response to decoding a particular sequence of instructions or receiving a special command to enter a new mode of execution.

The preferred embodiment identifies the beginning of an AIG after decoding of an instruction (**800**). If it is determined that a BEGIN_AIG instruction has been decoded (**805**) the processor enters a new MILLICODE MODE (**815**) and begins execution of millicode routines which have been written to support transactional memory execution. While the processor is in the MILLICODE MODE it has access to additional hardware state registers. One such state register indicates a new mode of execution, AIG MODE, which is used by the processor to control whether millicode is used to assist with the execution of other instructions associated with the support of transactional memory. In the current invention, MILLICODE MODE is used to execute complex instructions that would be difficult to execute in a single pass of the instruction pipeline of a typical Reduced Instruction Set Computing (RISC) system. The processor enters AIG MODE and then millicode is used to create the transaction tables that are required for the support of transactional memory. If the instruction to be decoded is not the beginning of an AIG then the instruction is executed as usual (**810**) and processing continues with the next instruction (**800**).

Prior to entering MILLICODE MODE, the architected state of the processor is saved in the same manner that a traditional Program Call is handled in the IBM Z-Series architecture. Any similar state saving mechanism may be used on any other processor architecture. The processor may return directly to this saved state if the AIG fails or may return to this state after a series of other error handling routines have been invoked. After the millicode associated with beginning a new AIG has been executed the processor exits MILLICODE MODE (**815**). The processor continues to be in AIG MODE at this point. While in AIG MODE, decoding of instructions continues (**820**). If a load or store is decoded (**825**) then special handling of these loads and stores is required (**840**). The processor enters MILLICODE SUBSET MODE to prepare for the execution of the processing steps required to track the transactional memory state. Entering the MILLICODE SUBSET MODE does not require as many processor cycles as MILLICODE MODE. The current invention uses a faster millicode setup path in the MILLICODE SUBSET MODE in order to minimize the time required to execute loads and stores that are part of an AIG. Additional processor state registers are added to enable the fast calculation of the table addresses required to track the state of the AIG. The steps required for the processing of a load inside of an AIG are described in FIG. **3**, those for a store inside of an AIG are described in FIG. **4**. Since the execution of loads and stores inside of an AIG is a very common occurrence, the MILLICODE SUBSET MODE is designed to provide very fast execution for their execution. After loads and stores in an AIG are processed the processor exits MILLICODE SUBSET MODE and it is determined (**850**) whether there has been an AIG failure as described in the descriptions of FIG. **3** and FIG. **4**. If there is a failure then special handlers are invoked (**860**). The actions of the special handlers for transaction failure may vary depending on the architecture of the system using the invention. Many methods of dealing with transaction failures have been described in the prior art, any of them could be used for this purpose. The possible methods include retrying the transaction from the beginning for a set number of attempts, waiting for the condition that caused the failure to change, calling specialized software routines to resolve conflicts among threads etc. These possible failure handlers may require the rollback of the AIG which caused the failure (**860**). If no failure was detected then it is determined whether the load or store was the last instruction of the AIG (**855**). If the instruction is the last instruction in the AIG then the AIG is committed (**865**) by entering MILLICODE MODE and using the process described in FIG. **5**. The processor then exits the AIG Mode and MILLICODE MODE and continues with the next sequential instruction after the AIG (**800**).

The decoding of instruction other than loads and stores inside of an AIG does not necessarily require special processing (**830**). If the instruction is not a load or store type instruction then it is determined whether the instruction is another BEGIN_AIG (**830**). Nested AIG instructions require special handling (**835**). The prior art contains many different ways of handling nested transactions. The current invention can be used to support any of them. The nested AIG handler for the preferred embodiment adds the instructions of the "inner" AIG to the "outer" AIG creating a single larger AIG. Other embodiments of the invention may provide different rules of processing for loads and stores that are part an inner nested transaction as well as special rules for the commitment and failure handling of inner nested transactions. If it is determined that the instruction is not a BEGIN_AIG then the instruction is executed (**845**) and it is next determined (**855**) whether the instruction is the last instruction of the AIG. The preferred embodiment uses the decoding of an END_AIG instruction to indicate that the last instruction of an AIG has been reached. If the last instruction of the AIG has been reached then the processor enters MILLICODE MODE, the AIG is committed to memory and the processor exits the AIG MODE of operation and then exits the MILLICODE MODE (**865**). If there is a failure then special handlers are invoked (**860**). The actions of the special handlers for transaction failure may vary depending on the architecture of the system using the invention. Many methods of dealing with transaction failures have been described in the prior art, any of them could be used for this purpose. The possible methods include retrying the transaction from the beginning for a set number of attempts, waiting for the condition that caused the failure to change, calling specialized software routines to resolve conflicts among threads etc. These possible failure handlers may require the rollback of the AIG which caused the failure (**860**). If no failure was detected then it is determined whether the load or store was the last instruction of the AIG (**855**). If the instruction is the last instruction in the AIG then the AIG is committed (**865**) using the process described in FIG. **5**. The processor then exits the AIG Mode and continues with the next sequential instruction after the AIG (**800**).

The decoding of instruction other than loads and stores inside of an AIG does not necessarily require special processing (**830**). If the instruction is not a load or store type instruction then it is determined whether the instruction is another BEGIN_AIG (**830**). Nested AIG instructions require special handling (**835**). The prior art contains many different ways of handling nested transactions. The current invention can be used to support any of them. The nested AIG handler for the preferred embodiment adds the instructions of the "inner" AIG to the "outer" AIG creating a single larger AIG. Other embodiments of the invention may provide different rules of processing for loads and stores that are part an inner nested transaction as well as special rules for the commitment and failure handling of inner nested transactions. If it is determined that the instruction is not a BEGIN_AIG then the instruction is executed (**845**) and it is next determined (**855**) whether the instruction is the last instruction of the AIG. The preferred embodiment uses the decoding of an END_AIG instruction to indicate that the last instruction of an AIG has been reached. If the last instruction of the AIG has been reached then the AIG is committed to memory and the processor exits the AIG MODE of operation (**865**).

The invention provides assist hardware that enables millicode to be invoked for quickly for certain cases, as illustrated by FIG. **9**.

Turning to FIG. **9**, it will be seen that the current invention uses an Address History Table (**950**) that is part of a multithreaded microprocessor (**900**) that has been designed for support of transactional memory and which uses millicode for the execution of complex instructions. The preferred embodiment uses a microprocessor that supports simultaneous multithreading (SMT) but other forms of multithreading such as hardware multithreading (HMT) may be used. The microprocessor in the preferred embodiment supports 4 simultaneous threads. Entries in the AHT are written by the load store units (**910**,**920**) over new interfaces (**930**,**940**) that carry command, control, address and data information. Command and control information may also be written into the AHT from the Instruction Execution Pipelines (**990**). The contents of the AHT may be copied or moved to the General Purpose Registers (**960**) of the microprocessor over the new interface (**980**) provided for that purpose. The contents may also be copied or moved to the Millicode Registers (**917**) via the Millicode Assist Setup Logic (**916**) over the interfaces

provided (**980**,**915**). The Millicode Registers are used by millicode instructions during the execution of millicode. The Millicode Assist Setup Logic (**916**) provides the ability to move the contents of the AHT (**950**) to positions in the millicode registers (**917**) that are well suited to fast execution using the native hardware instructions of the microprocessor. The Millicode Assist Setup Logic (**916**) may also shift and/or increment the contents of the AHT to provide additional speed of execution when running the millicode routines required to support transactional memory. The microprocessor (**900**) also contains new state registers for indication of when the microprocessor enters MILLICODE MODE (**917**) and AIG MODE (**918**). These states may be set or reset using the interfaces provided (**921**,**922**). The traditional use of the millicode mode state register (**917**) as part of the control logic of a millicoded microprocessor is well known. The millicode assist setup logic (**916**) of the current invention uses control information from the Instruction Execution Pipelines (**990**) and AHT (**950**), via the interfaces provided (**970**,**980**), to anticipate when the microprocessor will enter millimode for the execution of millicode instructions that are used to support transactional memory operations. This early indication allows for the early setup of the millicode registers (**917**) and quick execution of the required code.

The current invention uses the new AIG MODE state (**918**) to limit the millicode hardware setup for loads and stores to just the minimal requirements for the executions of loads and stores while AIG MODE is active. Traditional millicoded microprocessors perform a wide variety of hardware setup steps when moving from normal execution mode to MILLICODE MODE. These setup steps may require dozens of microprocessor clock cycles in some cases. One object of the current invention is focused on reducing this overhead for the common case of setup for the execution of loads and stores while the AIG MODE is active.

FIG. **10** illustrates more detail of the Address History Table (**1000**). In the preferred embodiment the Address History Table (AHT) is a hardware register file. The AHT could be implemented as an SRAM or DRAM array or any other suitable storage element. The AHT could also be implemented outside the microprocessor core or on another chip. The AHT could also be implemented as a software table in main memory. The AHT contains an "Entry Type" field (**1010**). A non-zero Entry Type is considered a "valid" entry. The Entry Type may be a specialized command or an "Address Record". In the case of an "Address Record" entry, the Access Type field (**1020**) will contain an indication of whether the address was from a store type access or a load type access. The "Address" field (**1030**) contains the real address of the storage access. The "Thread ID" field (**1040**) indicates the thread of execution that is processing an AIG and which caused the entry in the AHT to be created. The "Response" field (**1050**) is used to associate hardware responses to AHT entries. The responses may indicate that steps required during the processing of addresses stored in the AHT have been completed. The response field may be just a single bit for one response or may be multiple bits which are used to represent multiple responses or multiple stages if a series of responses are required.

FIG. **11** illustrates software tables built by the AIG Assist Thread. FIG. **11** illustrates one pseudo code version of the code that the current invention uses to manage the transaction tables associated with an AIG. The "pseudo code" shown in FIG. **11** is written in the style of the IBM PowerPC microprocessor architecture but the code can be translated into code for any other suitable microprocessor architecture. The current invention requires the pseudo code to be translated into millicode. The translation is done in advance of the runtime and the resulting code is recorded and stored with the other millicode used by the target system on tangible media for use by the target system. The current invention requires millicode for each of the transactional memory functions. These functions include creating and updating the transaction tables, committing transactions and controlling the rollback of transactions which fail. All of these functions can be described in the same manner as shown in FIG. **11** and translated into millicode.

The PowerPC architecture uses Special Purpose Registers (SPRs) for many purposes related to storing machine state in a microprocessor. The Address History Table of the current invention stores state information related to currently active AIGs. The "pseudo code" in FIG. **11** uses the SPR naming convention when referencing the AHT. The "pseudo code" uses the PowerPC style of referencing General Purpose Registers, i.e. R4. The current invention uses millicode that implements the same function using millicode registers. The setting of the PTRAN bit or bits associated with each load and store inside of an AIG is shown in the pseudo code example with the terms "write the correct bit".

The loading of registers and shifting of contents are examples of functions that the current invention performs using the millicode assist setup logic (**916**). The invention allows these operations to be done in by specialized hardware in parallel with the traditional millicode setup operations that are used to move the microprocessor from the normal state to the millicode active state. The use of hardware assists for simple functions and millicode for more complex functions allows the invention to provide a balance of speed and flexibility when implementing the complex operations required for the support of a transactional memory programming environment.

The flow diagrams and code depicted herein are just examples. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

While the preferred embodiment to the invention has been described, it will be understood that those skilled in the art, both now and in the future, may make various improvements and enhancements which fall within the scope of the claims which follow. These claims should be construed to maintain the proper protection for the invention first described.

What is claimed is:

1. A computing system with transactional memory, comprising:

a multithreaded microprocessor configured to support the transactional memory, the microprocessor configured to execute complex instructions using millicode by a millicode mode and an atomic instruction group (AIG) mode, the microprocessor comprising:

an address history table (AHT) coupled to load and store units, the load and store units configured to write information entries in the AHT, the AHT coupled to instruction execution pipelines and configured to write information into the AHT from the instruction execution pipelines;

millicode mode state registers and AIG state registers coupled to the instruction execution pipelines;

general purpose registers coupled to millicode assist setup logic, the millicode assist setup logic coupled to millicode registers, the general purpose registers coupled to the AHT and configured to enable contents of the AHT to

be copied or moved to the general purpose registers of the microprocessor and also copied or moved to the millicode registers by the millicode assist setup logic over interfaces provided for the millicode registers, the millicode registers configured to be utilized by millicode instructions based on the millicode being executed.

2. The computer system according to claim **1** wherein the millicode is invoked by the millicode assist setup logic to move the contents of the AHT to positions in the millicode registers that are configured to execute based on native hardware instructions of the microprocessor.

3. The computer system according to claim **2** wherein the millicode assist setup logic is configured to shift, increment, or shift and increment the contents of the AHT to provide additional speed of execution based on running the millicode routines required to support transactional memory.

4. The computer system according to claim **2** wherein the millicode mode state registers and atomic instruction group (AIG) state registers are configured to indicate the microprocessor entering millicode mode and AIG mode respectively, and wherein the states are configured to be set or reset based on the interfaces provided.

5. The computer system according to claim **1** wherein the millicode assist setup logic is configured to anticipate the microprocessor entering millicode mode to execute millicode instructions used to support transactional memory operations, the millicode assist setup logic configured to use control information from the instruction execution pipeline and the AHT to anticipate the microprocessor entering millicode mode.

6. The computer system according to claim **1** wherein the AIG mode state registers limits the millicode hardware setup for loads and stores to minimal requirements for the executions of loads and stores based on the AIG mode being active.

7. The computer system according to claim **1** wherein the AHT contains an entry type field for which a non-zero entry type is considered a valid entry.

8. The computer system according to claim **7** wherein an entry type for the entry type field may be a specialized command or an Address Record, and, based on an Address Record entry, an access type field contains an indication of whether the address was from a store type access or a load type access.

9. The computer system according to claim **8** wherein an Address field contains a real address of a storage access, and a Thread ID field indicates a thread of execution that is processing an AIG and which caused an entry in the AHT to be created.

10. The computer system according to claim **9** wherein a Response field is configured to associate hardware responses to AHT entries, and wherein the responses indicate that steps required during the processing of addresses stored in the AHT have been completed.

11. The computer system according to claim **10** wherein the response field is either a single bit for one response or multiple bits configured to represent multiple responses or multiple stages based on a series of responses being required.

* * * * *